



Angular

Expliqué par

David Gilson

<https://github.com/gilsday>

Pourquoi du JavaScript ?



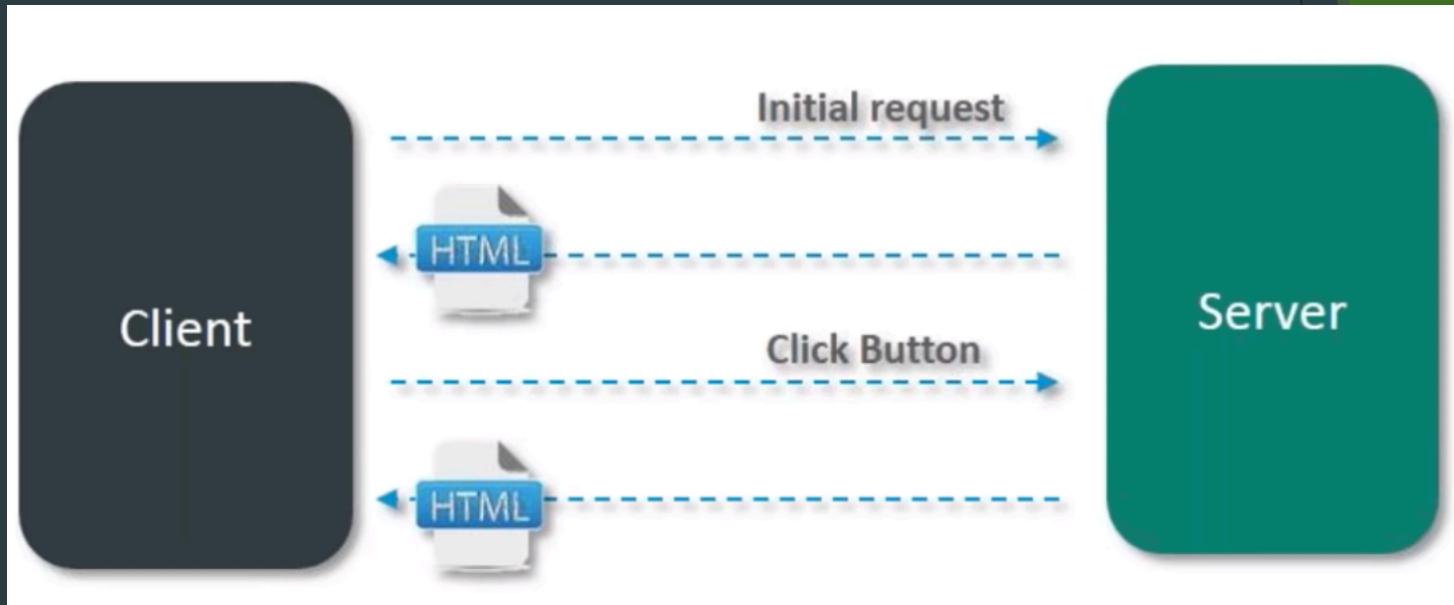
- ▶ Complètement indépendant de toute plateforme
 - ▶ Browser
 - ▶ Server
 - ▶ IBM LoopBack
 - ▶ Sails
 - ▶ Desktop
 - ▶ Electron
 - ▶ Mobile
 - ▶ Cordova
 - ▶ NativeScript
 - ▶ IOT
 - ▶ Intel XDK
 - ▶ Johnny-five
 - ▶ Cylon
 - ▶ IA
 - ▶ TensorFlow.js

Mais encore

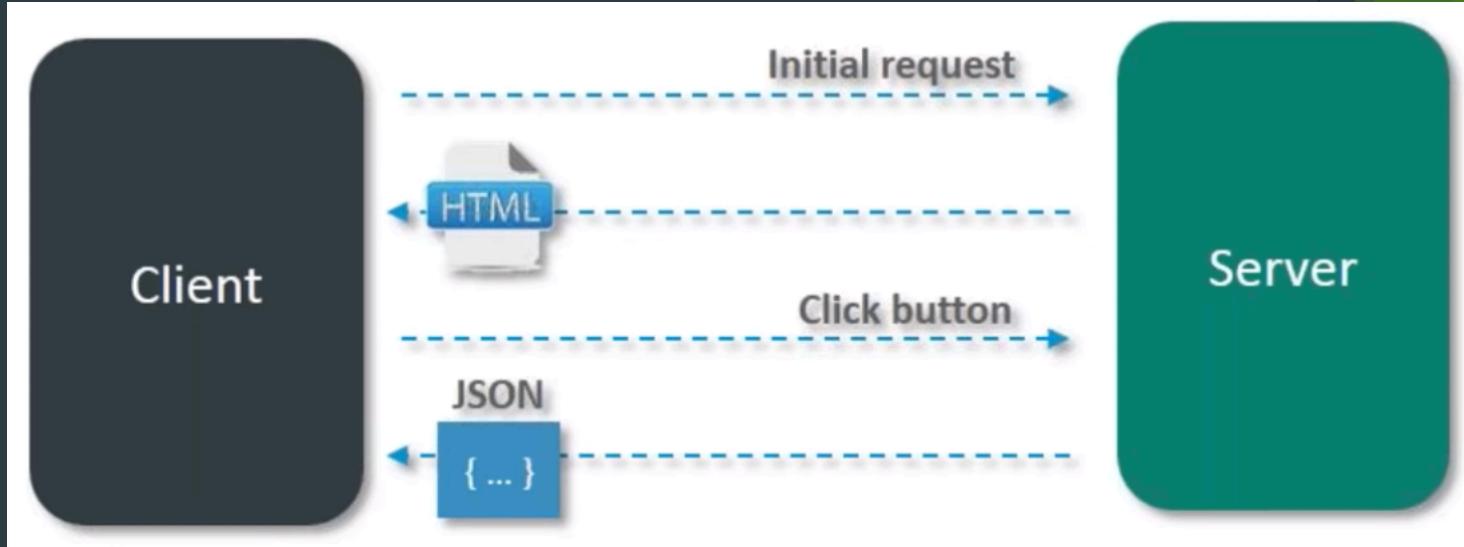


- ▶ SPA (Single Page Application)
 - ▶ Meilleur expérience utilisateur car plus fluide

Application traditionnelle



Single Page Application



NodeJS ?



- ▶ NodeJS est une technologie libre qui offre la possibilité d'exécuter du JavaScript en dehors d'un navigateur. Il est donc utilisé pour :
 - ▶ Crée des applications serveur.
 - ▶ Créer des scripts de build/test/...
- ▶ NodeJS intègre un puissant gestionnaire de package (npm).
- ▶ Basé sur le moteur JS V8, créé par Google en 2008 et qui a révolutionné les possibilités du JavaScript
- ▶ NodeJs ne nécessite pas de serveur http comme Apache, Tomcat ou IIS, il contient une bibliothèque HTTP (express).



TypeScript ?



- ▶ TypeScript est un langage de programmation libre et open-source développé par Microsoft qui a pour but de simplifier la création d'applications web.
- ▶ C'est un sur-ensemble de JavaScript (c'est-à-dire que tout code JavaScript correct peut être utilisé avec TypeScript).
- ▶ Le code TypeScript est transcomplié en JavaScript, pouvant ainsi être interprété par n'importe quel navigateur web ou moteur JavaScript.
- ▶ TypeScript permet un typage statique optionnel des variables et des fonctions, la création de classes et d'interfaces, l'import de modules, tout en conservant l'approche non-constrictrice de JavaScript.



Immutabilité

▶ Mutation

- ▶ Presque tout en JavaScript est mutable. Cela veut dire que l'on peut modifier un objet sans modifier sa référence.
- ▶ La mutation est très rapide
- ▶ Exemple d'éléments mutables: Function, Object, Array

▶ Immuabilité

- ▶ Changer la référence permet de pouvoir savoir plus rapidement que quelque chose a changé dans un objet (pas besoin de deepChecking). Meilleur pour les Input Angular.
- ▶ La manipulation d'objets de façon immutable est plus lente
- ▶ Exemple d'éléments immuables: Number, String, Boolean

Immutabilité



- ▶ Exemple modification par mutation

```
const testValue = {test: 'test'}; // ref = 1
testValue.test = 'test2';
// resultat: {test: 'test2'} => ref = 1
```

- ▶ Exemple modification immutable

```
let testValue = {test: 'test'}; // ref = 1
testValue = {...testValue, test: 'test2'};
// resultat: {test: 'test2'} => ref = 2
```

- ▶ Attention le Spread Operator (...) ne fait pas de copie profonde (deep copy)

Lambda vs Function



- ▶ **Function** : **this** fait référence à l'objet qui a fait **appel** à la fonction

```
function() {}
```

- ▶ **Lambda** (fonction fléchée) : **this** fait référence à l'objet où elle est **créée**

```
() => {}
```

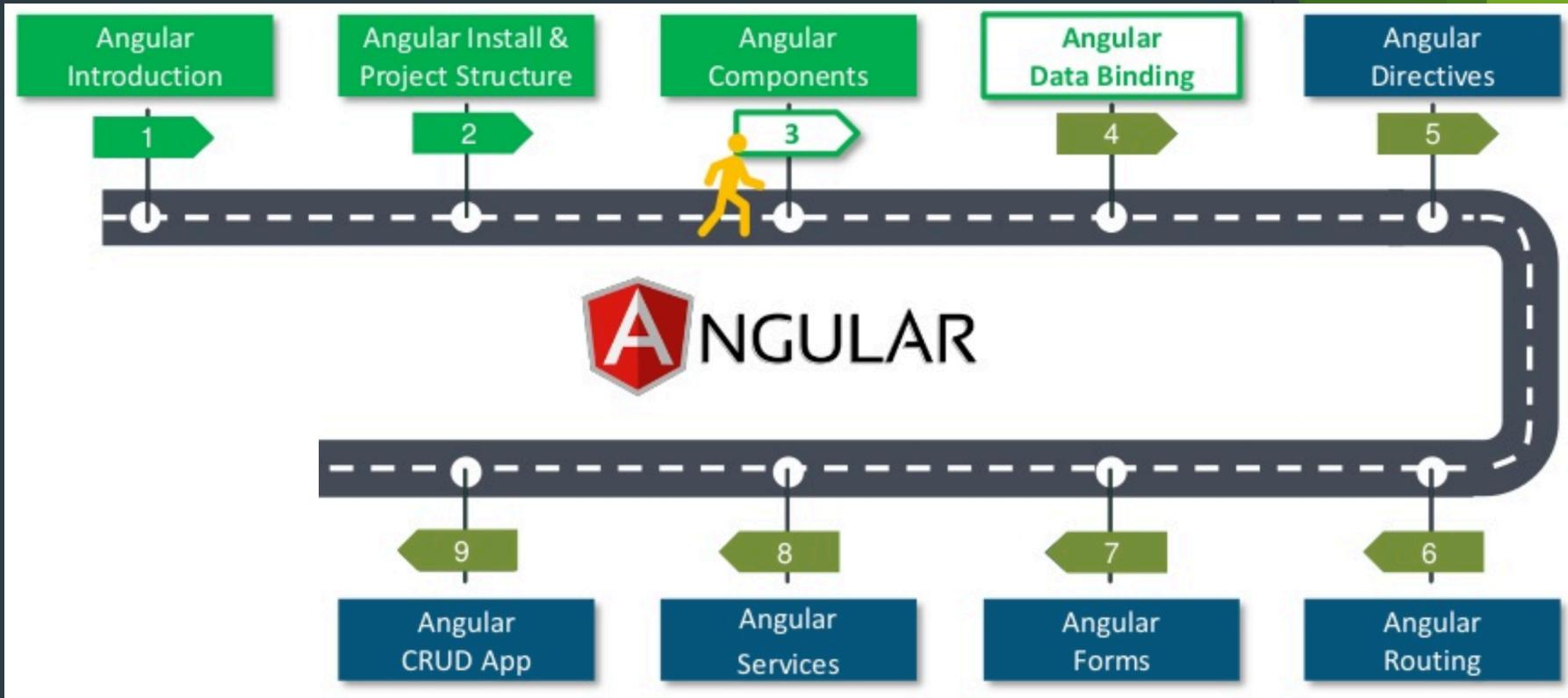


C'est quoi Angular ?

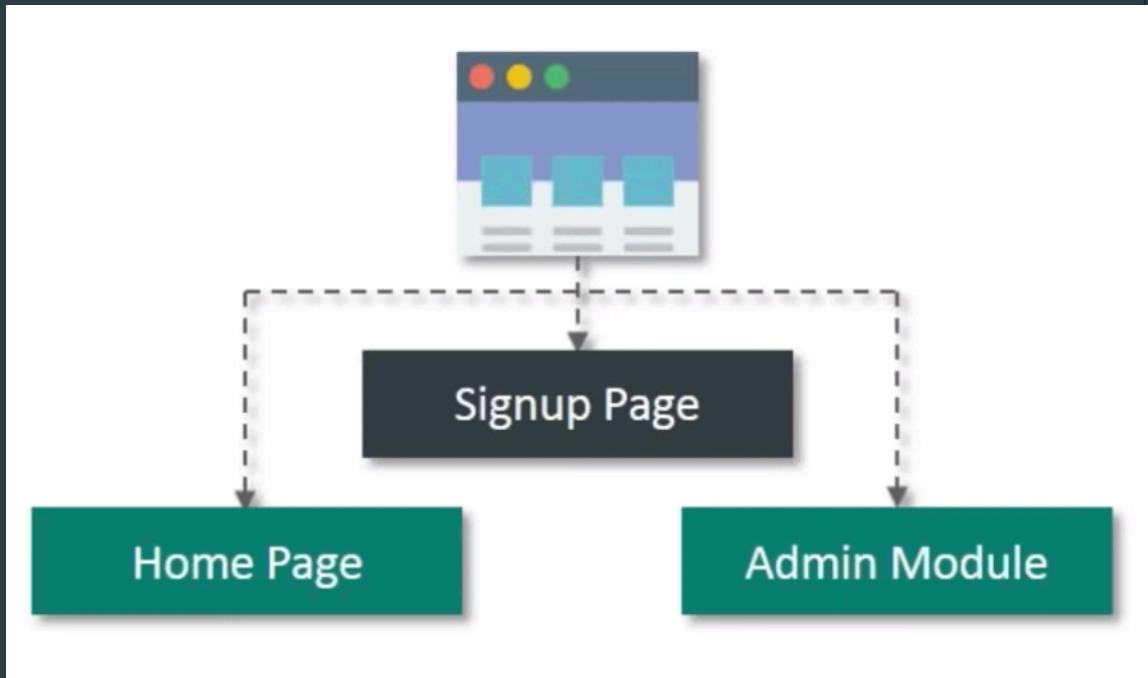


C'est quoi Angular ?

- ▶ Un Framework JavaScript pour faire des SPA
- ▶ Ecrit en TypeScript
- ▶ Crée par Google et Microsoft
- ▶ Architecture basée sur les composants
 - ▶ Inspiré par le concept de WebComponent
 - ▶ https://developer.mozilla.org/fr/docs/Web/Web_Components

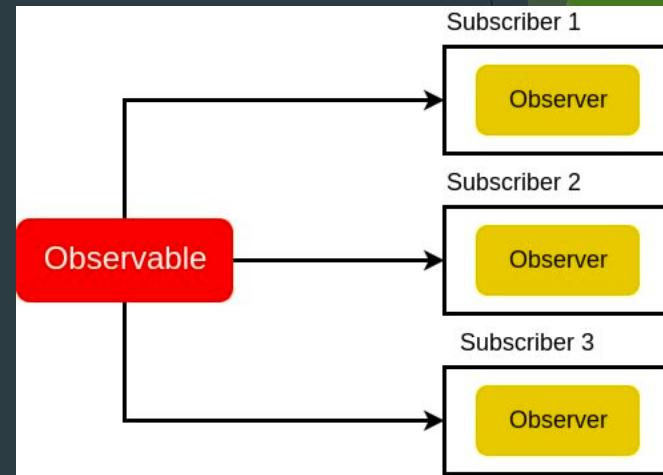
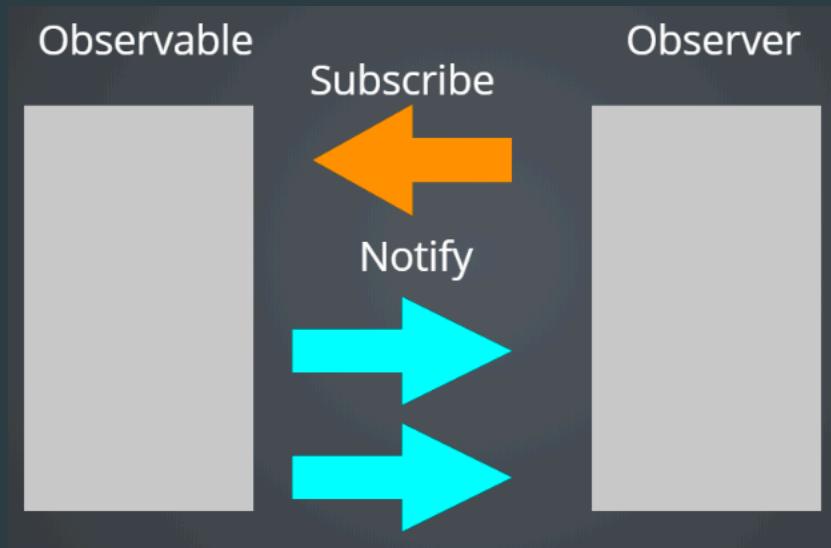


Modularité



Reactive programming

- ▶ Asynchrone Asynchrone Asynchrone...
- ▶ Observer pattern



Exercice

RXJS

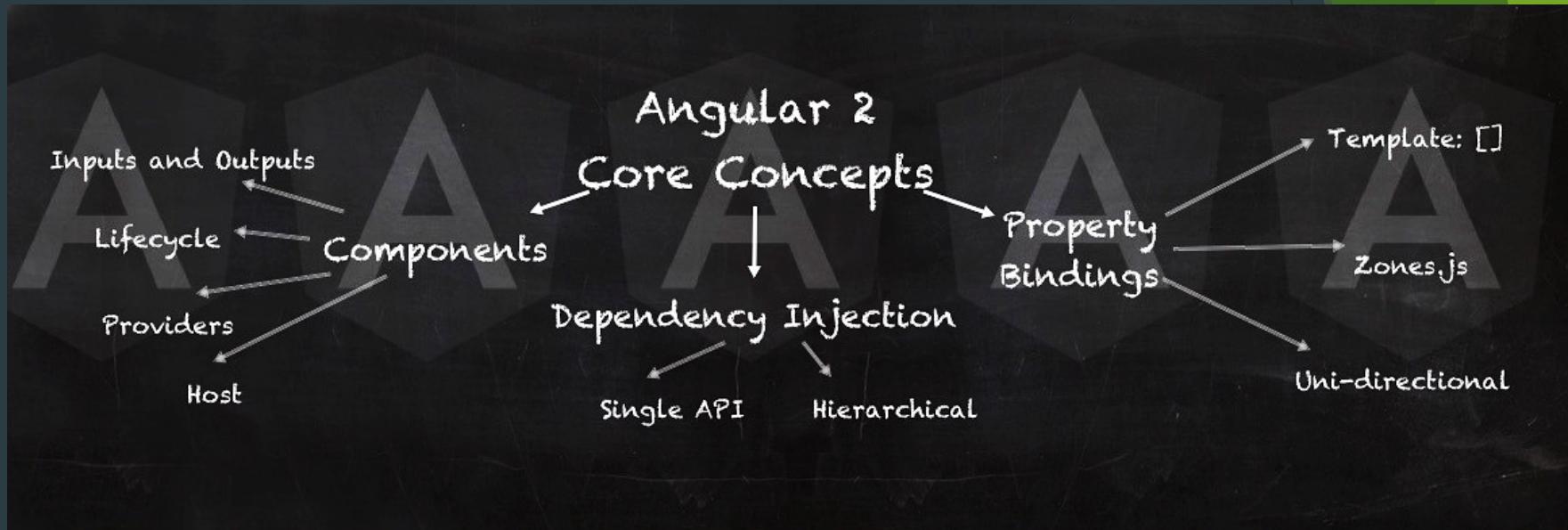
- ▶ Créons notre premier observable



Modules de base

- ▶ **@angular/core** : la base d'Angular (requis)
- ▶ **@angular/common** : directives et services commun (requis)
 - ▶ **@angular/common/http** : services permettant des appels http
- ▶ **@angular/router** : gestion des routes
- ▶ **@angular/forms** : directives et services pour la création de formulaires
- ▶ ...

Concepts





Installation et structure de projet



Outils

- ▶ NodeJS (<https://nodejs.org>) LTS+
- ▶ Visual Studio Code (<https://code.visualstudio.com>)
 - ▶ Extensions :
 - ▶ Angular Extension Pack
 - ▶ Angular Files
- ▶ Angular Cli ((sudo) npm install -g @angular/cli)
- ▶ Chrome (<https://www.google.com/intl/fr/chrome/browser>)
 - ▶ Extension :
 - ▶ Augury : <https://chrome.google.com/webstore/detail/augury/elgalmkoelokbchhkacckoklkejnhcd>

Exercice

CLI

- ▶ Cr ons un projet ensemble et regardons le r ultat

Module

- ▶ Un module peut contenir
 - ▶ Components
 - ▶ Services
 - ▶ Pipes
 - ▶ Guards
 - ▶ D'autres modules

Module

- ▶ **providers** : Un provider est la façon d'instancier un injectable (service/guard). Lister les services et les guards ici pour qu'ils deviennent injectables (utilisables).
- ▶ **declarations** : Déclarez ici les différents components/directives/pipes afin de pouvoir utiliser leurs selector/name.
- ▶ **entryComponents** : Permet de déclarer au transpileur que nous avons besoin de components qu'il pense non utilisés. Cela arrive quand on fait une instantiation dynamique d'un component. (Il est également nécessaire de le déclarer)
- ▶ **imports** : Liste des modules à importer (aussi bien internes que les librairies).
- ▶ **exports** : Liste des components/pipes à mettre à disposition d'autres modules.
- ▶ **bootstrap** : Components indépendants.



Module

- ▶ Pourquoi faire plusieurs modules ?
 - ▶ Réutilisabilité interne à l'application
 - ▶ Réutilisabilité entre applications (librairies)
 - ▶ Performance (LazyLoad)



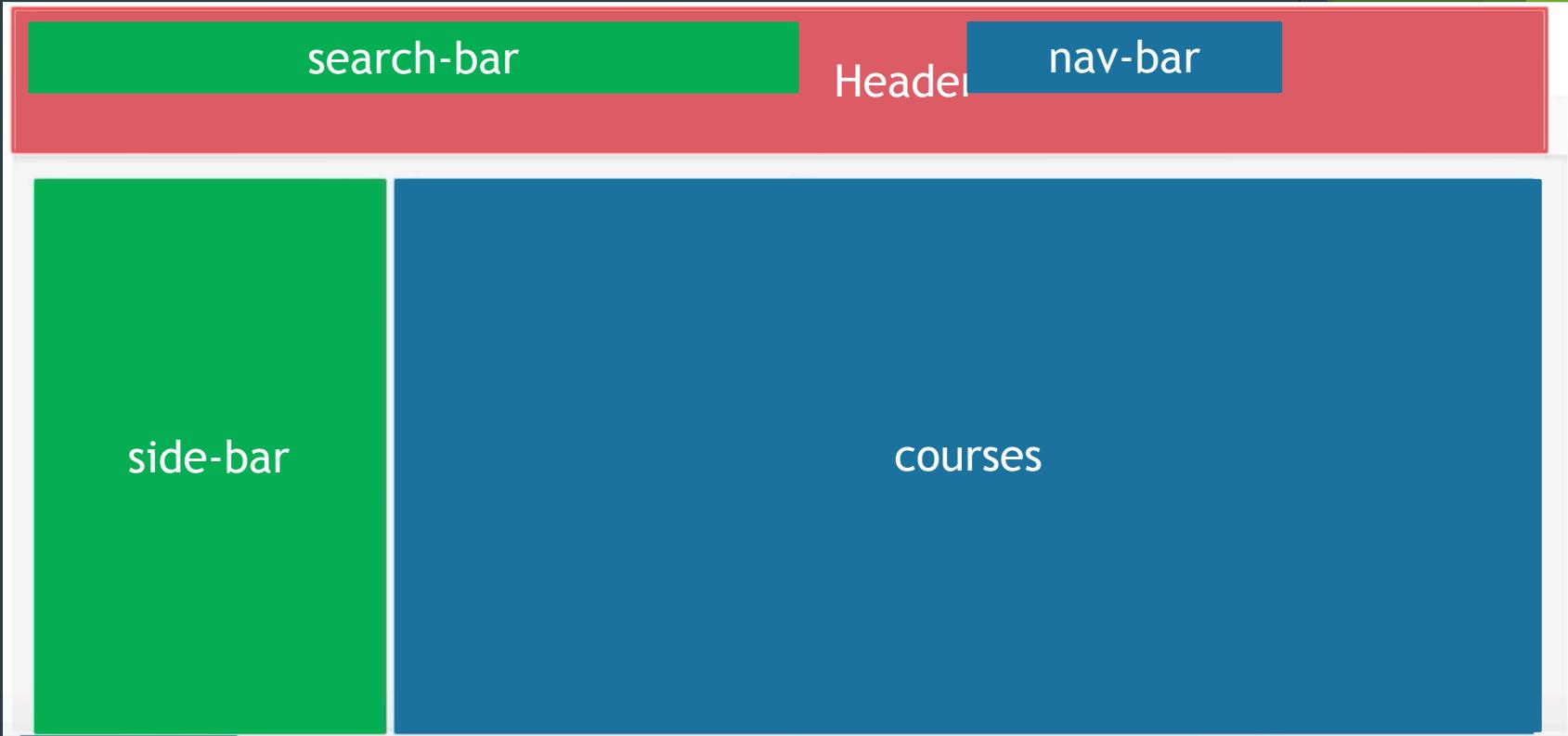
Component



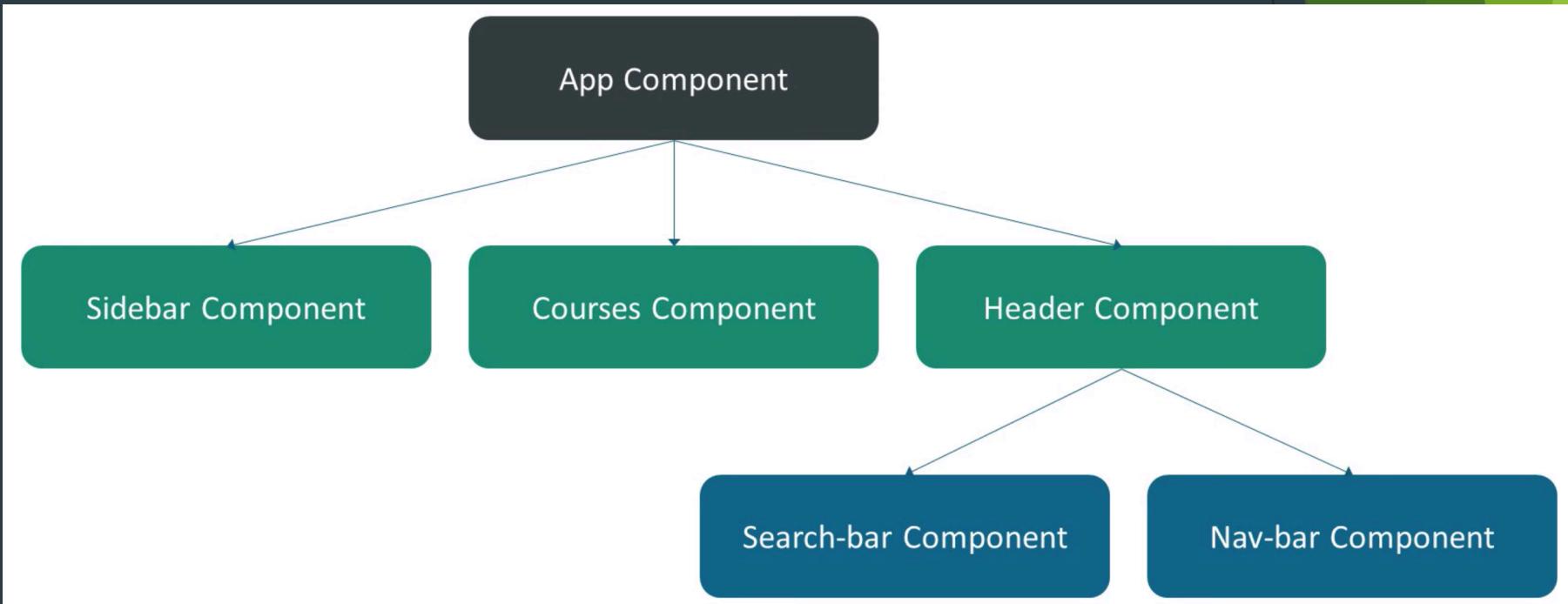
Composant

- ▶ Créer un composant, c'est créer un nouveau tag HTML
 - ▶ exemple: <my-component></my-component>
- ▶ Tout ce qui est visible est un composant
- ▶ Composé d'un fichier...
 - ▶ HTML
 - ▶ TypeScript
 - ▶ Une classe
 - ▶ Un décorateur
 - ▶ CSS/SCSS
- ▶ Identifié à l'aide d'un selecteur
 - ▶ exemple: 'my-component'

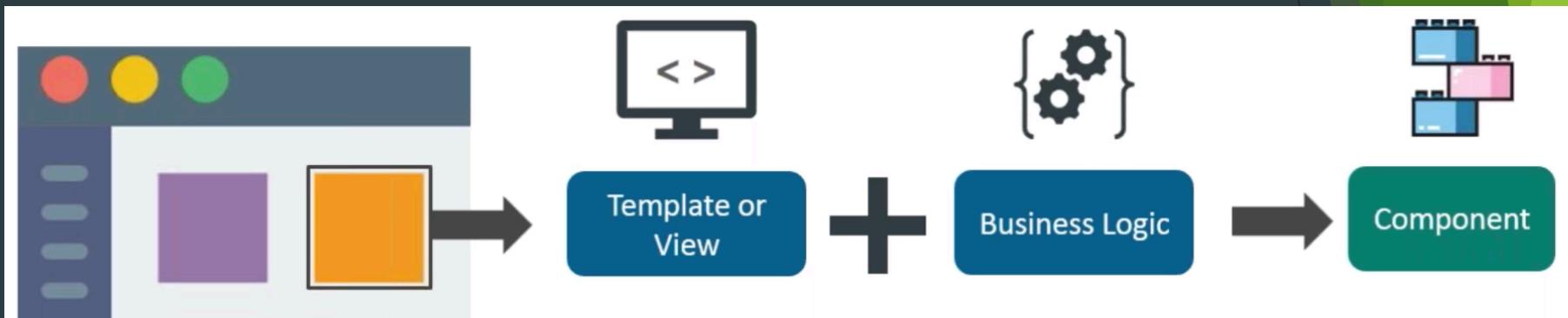
Découpe



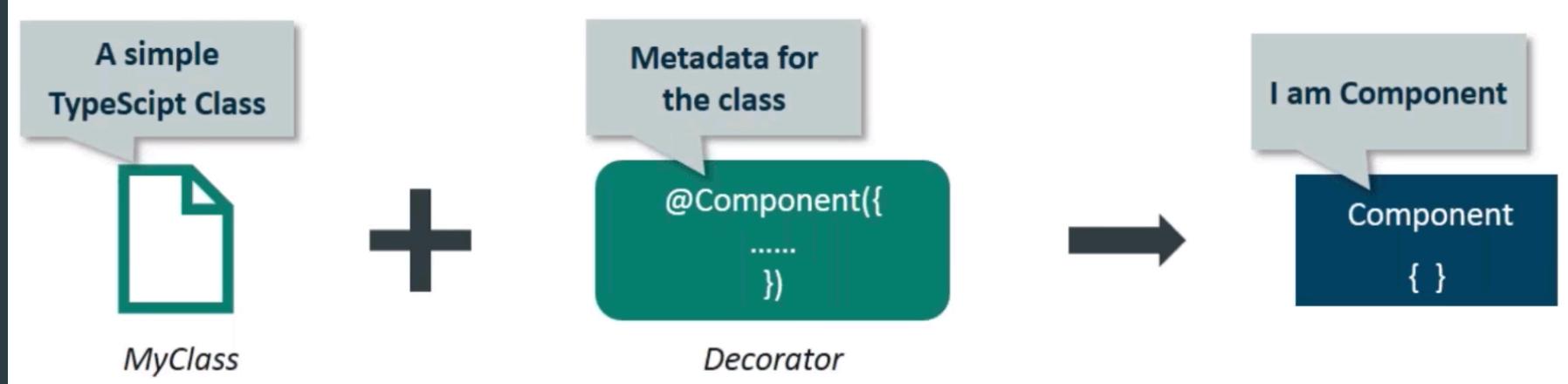
Découpe



Composant



Comment écrire un composant ?



Syntaxe composant

```
export class AppComponent{  
  constructor() { }  
}
```

A simple
TypeScript Class

```
@Component({  
  selector: 'app-app',  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.css']  
})
```

Metadata for
the class

```
import { Component } from '@angular/core';  
  
@Component({  
  selector: 'app-app',  
  templateUrl: './app.component.html',  
  styleUrls: ['./app.component.css']  
})  
  
export class AppComponent{  
  constructor() { }  
}
```

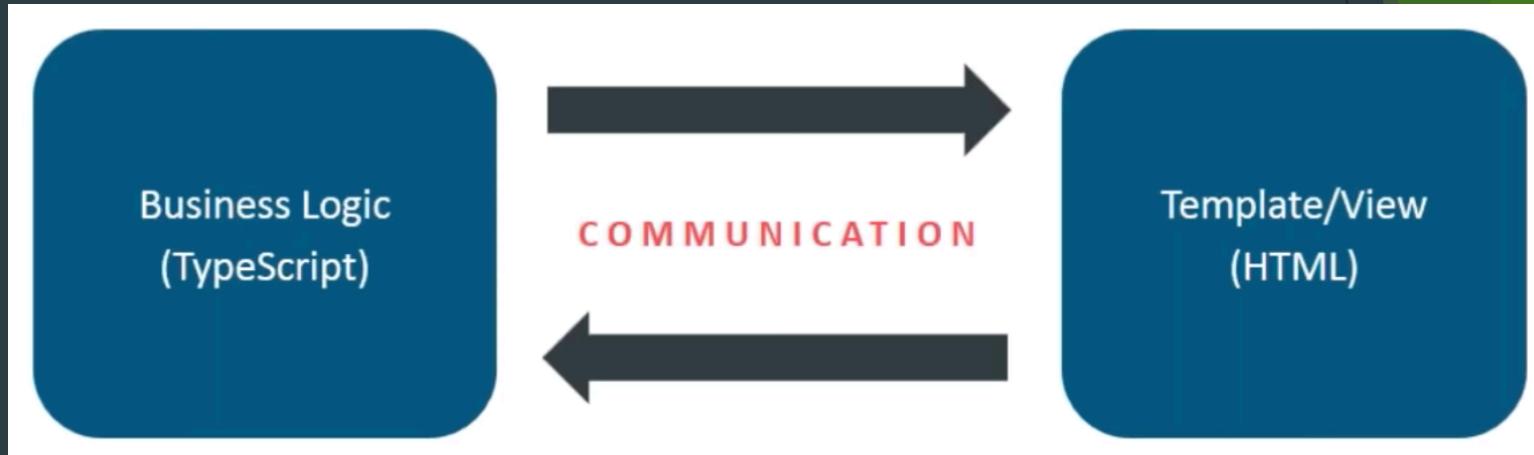
Exercice

Component

- ▶ Cr ons notre premier composant

Communication entre composants (binding)

- ▶ Le binding est un pont entre la vue (View) et la logique (View Model)



- ▶ MVVM

Communication entre composants (binding)

TYPES OF DATA BINDING

Data binding plays an important role in communication between
a template and its component

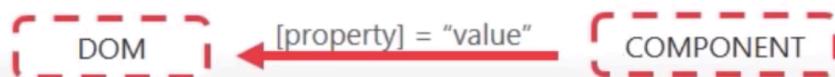
INTERPOLATION

01



PROPERTY BINDING

02



EVENT BINDING

03



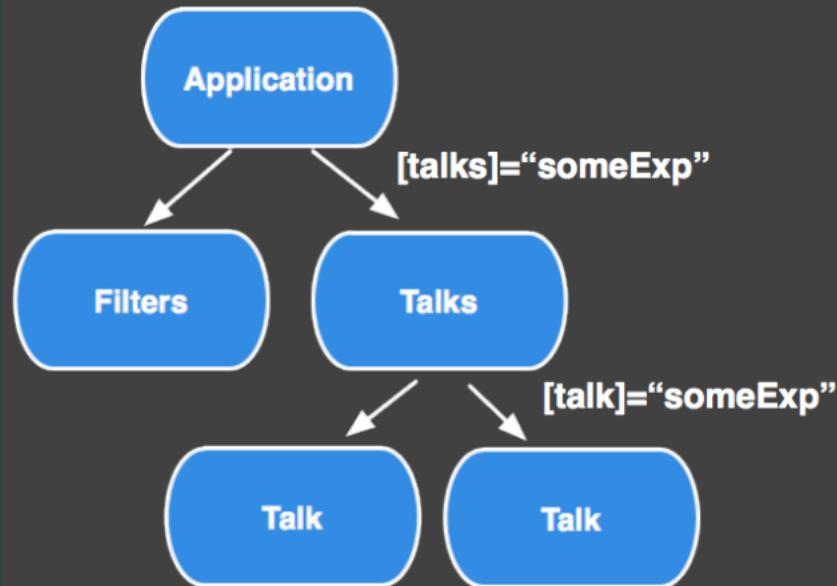
2 WAY DATA BINDING

04

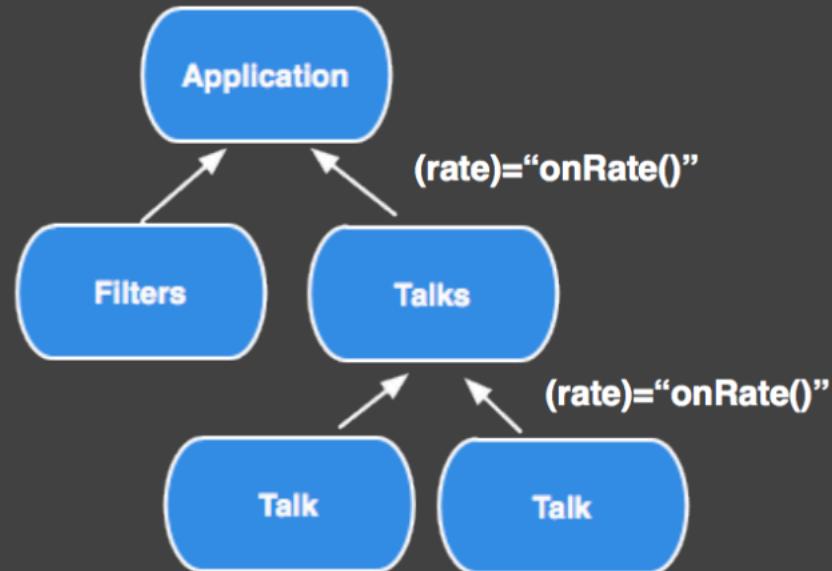


Communication entre composants (binding)

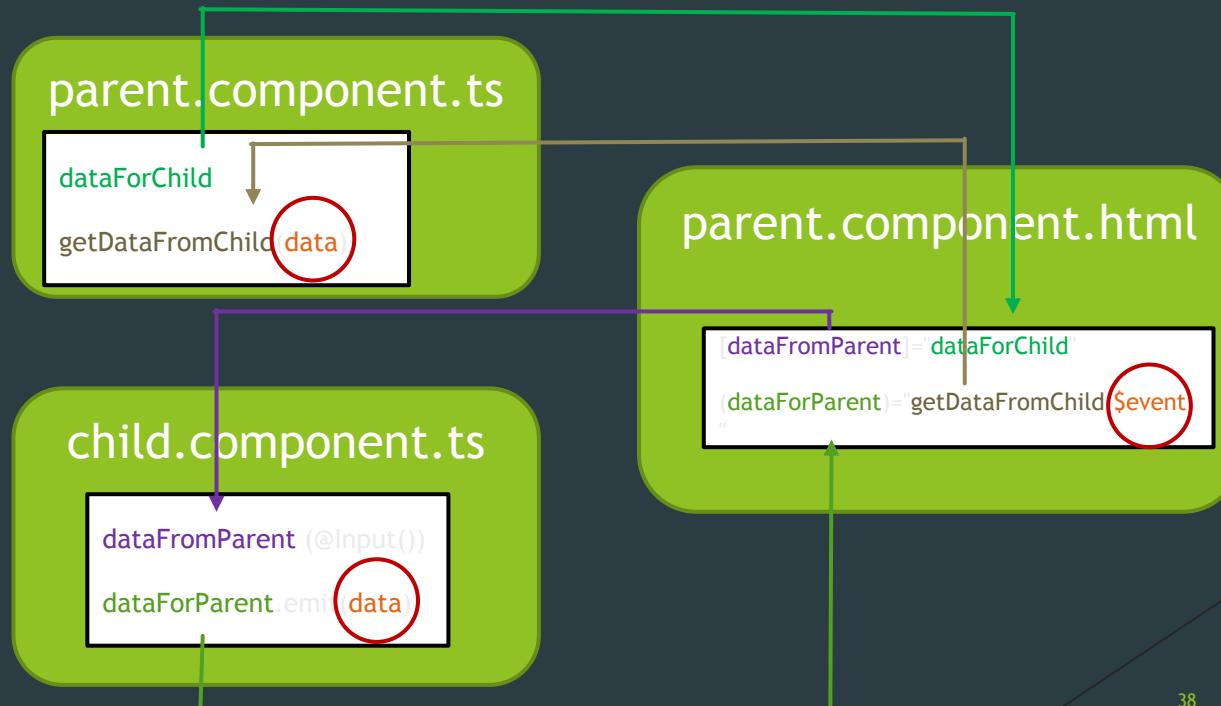
Parent -> Child



Child -> Parent



Fonctionnement des bindings





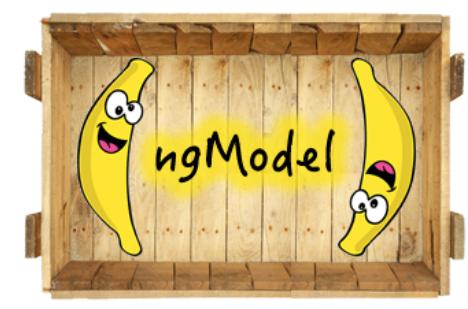
Syntaxe composant

```
@Component({
  selector: 'app-talks',
  template: '<h1>{{title}}</h1>' ← interpolation
})
class TalksComponent {
  @Input() talks: Talks[];
  @Output() rate = new EventEmitter<number>();
  title = 'Coucou';
}

<app-talks
  [talks]="myTalks"  (rate)="onRateChange($event)"
></app-talks>
```

Syntaxe composant

- ▶ [] = @Input()
- ▶ () = @Output()
- ▶ [()] = @Input() + @Output()
 - ▶ L'output doit avoir le même nom que l'input suivit de "change"
 - ▶ Input: data
 - ▶ Output: dataChange



Exercice

Component

- ▶ Cr ons un second composant et testons tous les types de binding

Cycle de vie

constructor

A l'appel de « **new** » (toute première fonction du cycle de vie)

ngOnChanges

A chaque changement de valeur d'un ou plusieurs **inputs**

ngOnInit

Après que le composant soit **initialisé** (après le premier ngOnChanges)

ngDoCheck

A chaque fois que Angular recalcule la vue

ngAfterContentInit

Après le premier check du contenu à appliquer à la vue

ngAfterContentChecked

Après chaque check du contenu à appliquer à la vue

ngAfterViewInit

Après que le **DOM** soit initialisé

ngAfterViewChecked

Après chaque écriture du DOM

ngOnDestroy

Avant la **destruction** du composant



Directives de base

```
<div *ngIf="name">{{name}}</div>
```

```
<div *ngFor="let element of elements">{{element}}</div>
```

```
<div [ngSwitch]="status">
  <span [ngSwitchCase]=""ok">OK</span>
  <span [ngSwitchCase]=""ko">KO</span>
  <span ngSwitchDefault>NO< span>
</div>
```



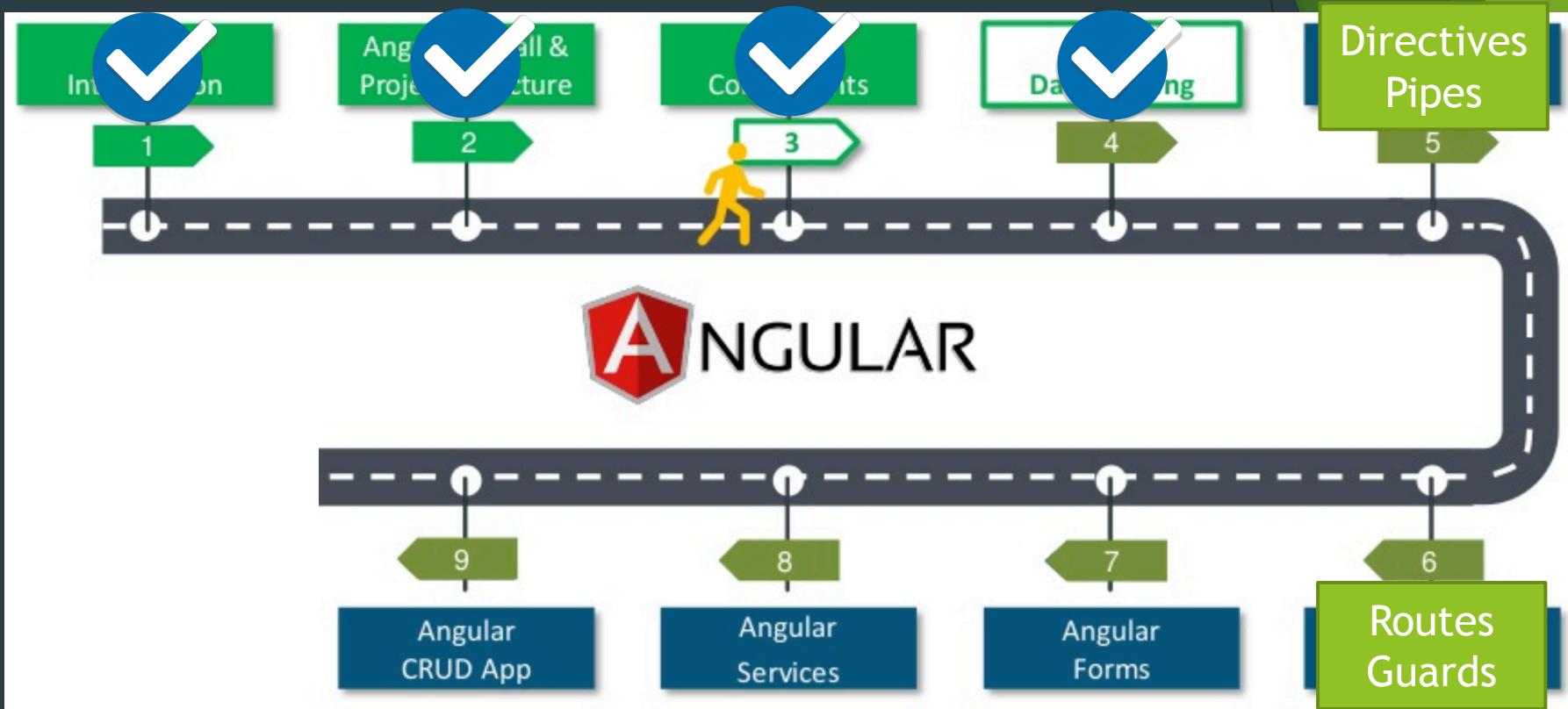
Evènements

- ▶ `element.addEventListener("click", myScript);`
- ▶ Angular fournit également un binding pour tous les évènements JavaScript
- ▶ `<element (click)="myScript($event)"></element>`

Exercice

Component

- ▶ Cr ons un second composant et testons tous les types de binding





Directive

Manipulation de l'HTML

- ▶ **Composant**
 - ▶ Directive avec vue (HTML/CSS)
- ▶ **Directive structurelle**
 - ▶ Gère si un élément apparaît ou non dans le DOM (exemple: *ngIf)
- ▶ **Directive attribut**
 - ▶ Change l'apparence ou le comportement d'un élément existant (exemple: required)

Directive structurelle

Built-in

- ▶ **NgIf**
 - ▶ Ajouter ou enlever un élément dans le DOM
- ▶ **NgFor**
 - ▶ Répéter un élément du DOM en itérant sur un tableau
- ▶ **NgSwitch**
 - ▶ Choisir l'élément à ajouter dans le DOM à l'aide d'une condition switch

Exercice

Directive

- ▶ Testons les 3 types de directives

Directive structurelle

Custom

- ▶ Mettre l'étoile devant le nom de la directive nous permet d'avoir accès au ViewContainerRef et au TemplateRef
 - ▶ `ViewContainerRef` est une référence vers un endroit du DOM où l'on peut ajouter un component ou un template.
 - ▶ Il est donc possible de vider le ViewContainerRef: `clear()`
 - ▶ Ou d'ajouter un template dans: `createEmbeddedView(this.templateRef)`
 - ▶ `TemplateRef` est le template sur lequel est placée la directive.



Directive structurelle

Custom

```
@Directive({
  selector: '[myNgIf]'
})
export class MyNgIfDirective implements OnChanges {

  @Input() myNgIf: any;

  constructor(
    private viewContainer: ViewContainerRef,
    private template: TemplateRef<Object>
  ) {}

  ngOnChange() {
    if (this.myNgIf) {
      this.viewContainer.createEmbeddedView(this.template);
    } else {
      this.viewContainer.clear();
    }
  }
}
```

Exercice

Directive

- ▶ Créons et utilisons notre propre version du nglf
 - ▶ Différence avec la version « built-in » : Nous ne voulons pas que 0 ou un string vide cache notre élément



Directive attribut

Built-in

- ▶ **NgStyle**
 - ▶ Ajouter du style CSS « inline » à un élément
- ▶ **NgClass**
 - ▶ Ajouter des classes CSS à un élément
- ▶ **NgModel**
 - ▶ Binding two-way standardisé



Directive attribut

Built-in

```
[ngClass] = "{'ma-classe': maCondition}"
```

Exercice

Directive

- ▶ Testons NgClass



Directive attribut

Custom

```
@Directive({
  selector: '[validPlateNumber]',
  providers: [
    {provide: NG_VALIDATORS, useExisting: PlateNumberValidator, multi: true }
  ]
})

export class PlateNumberValidator implements Validator {}
```



Pipe

Pipe

- ▶ Un pipe est une couche d'interprétation/manipulation de données
- ▶ Utilisé dans l'HTML
- ▶ Plusieurs utilités:
 - ▶ Permettre de déterminer comment afficher un objet
 - ▶ Filtrer les données d'une boucle (ngFor)
- ▶ Chainable
 - ▶ Appliquer plusieurs pipes

Pipe

```
<p>The hero's birthday is {{ birthday | date:"dd/MM/yy" }}</p>
```

```
<p>The hero's birthday is {{ birthday | date:"dd/MMMM/yy" | uppercase }}</p>
```

```
<div *ngFor="let item of (list | myFilter:valueToFilter)">  
  {{item | json}}  
</div>
```



Pipe

Built-in

- ▶ Date
- ▶ Uppercase
- ▶ Lowercase
- ▶ Currency
- ▶ Percent
- ▶ Json
- ▶ KeyValue
- ▶ ...

Pipe

- ▶ Deux types de pipe:
 - ▶ Pure
 - ▶ N'a accès qu'aux données fournies en input pour faire son traitement
 - ▶ Exemple:UpperCase
 - ▶ Impure
 - ▶ Peu faire appel à des éléments extérieurs (non fourni par l'utilisateur du pipe)
 - ▶ Exemple: Date (a besoin d'aller chercher ailleurs la langue à appliquer)

Exercice

Pipe

- ▶ Testons le pipe Date et le pipe JSON



Pipe

Custom

```
@Pipe({  
    name: 'myFilter'  
})  
export class MyFilterPipe  
implements PipeTransform {  
    transform(value: any, args?: any): any {}  
}
```

Exercice

Pipe

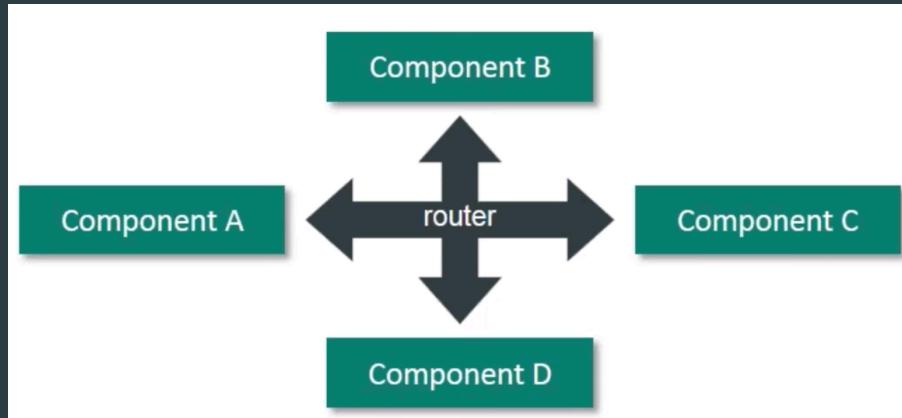
- ▶ Créons un pipe qui va trier un tableau dans l'ordre alphabétique



Routing

Routing

- ▶ La gestion d'accès aux différentes pages se fait via l'URL
 - ▶ Comme n'importe quels autres sites
 - ▶ Mais sans faire l'allez-retour vers le serveur
- ▶ Pour savoir quelle page afficher pour quelle URL, il faut mettre en place le routing.





Routing

```
export const routes: Routes = [
  { path: '', component: ParentComponent, children: [
    { path: '', redirectTo: '/home', pathMatch: 'full' },
    { path: 'home', loadChildren: './home/home.module#HomeModule',
      canActivate: [AuthGuard] },
    { path: 'login', loadChildren: './login/login.module#LoginModule' },
  ] },
  { path: '**', redirectTo: '/home' }
];

RouterModule.forRoot(routes)
```

Routing

- ▶ Mais où se retrouve le composant choisi par le router ?

```
<router-outlet></router-outlet>
```

Exercice

Routing

- ▶ Cr ons une autre page accessible depuis l'URL « /contact »
 - ▶ Cela implique de d placer le contenu de « app-component » g alement dans une page « home »

Guard

Guard

- ▶ Un Guard permet de gérer l'accès aux routes
 - ▶ Sécuriser la navigation
- ▶ Deux types de Guards:
 - ▶ **OnActivate**: Peut-on accéder à la page ?
 - ▶ **OnDeactivate**: Peut-on quitter la page ?



Guard

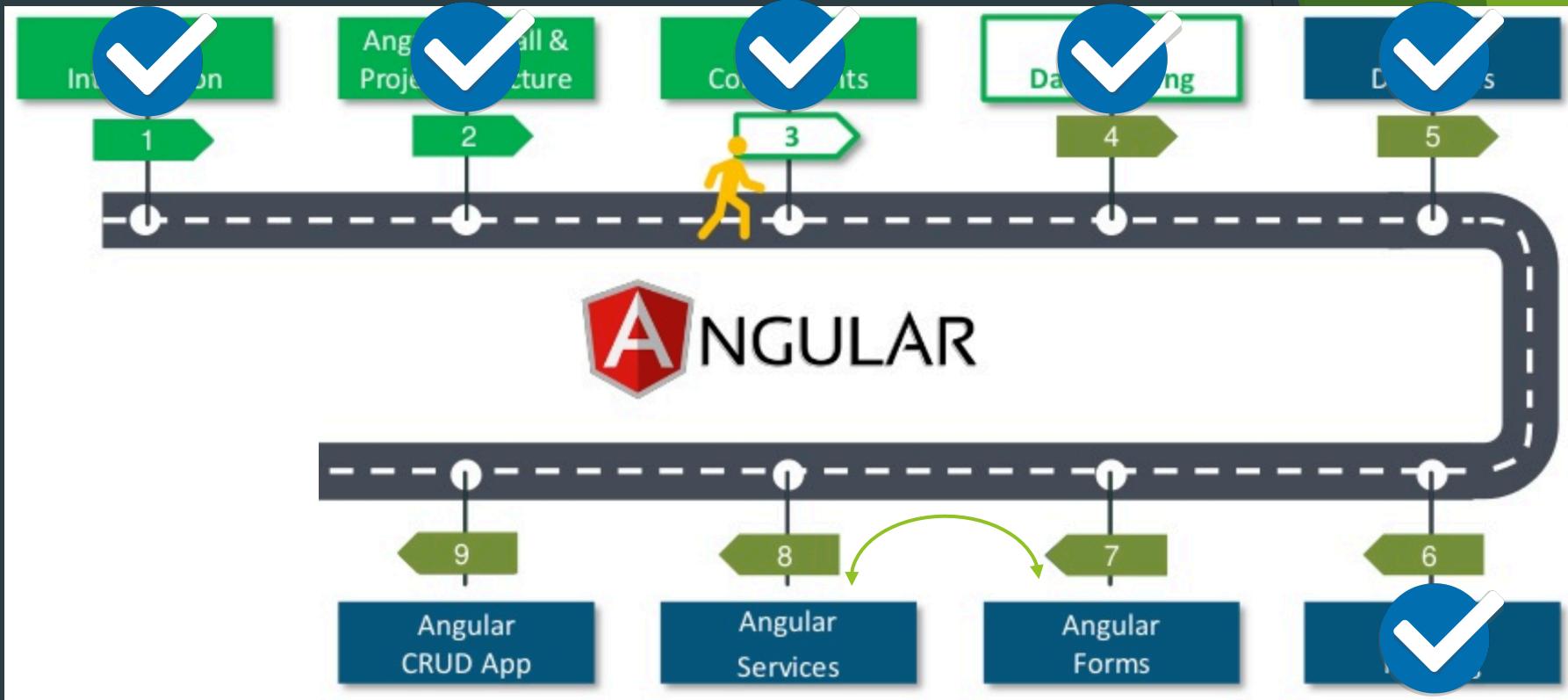
```
@Injectable()
export class NeedAuthGuard implements CanActivate {
  constructor(
    private router: Router,
    private authenticationService: AuthenticationService
  ) {}

  canActivate(route: ActivatedRouteSnapshot): Observable<boolean> {
    return this.authenticationService.isConnected().pipe(map(isConnected => {
      if (!isConnected) {
        this.router.navigate(['/login']);
      }
      return isConnected;
    }));
  }
}
```

Exercice

Guard

- ▶ Bloquons l'accès à la page « contact » tant que le LocalStorage « contact-access » n'existe pas.



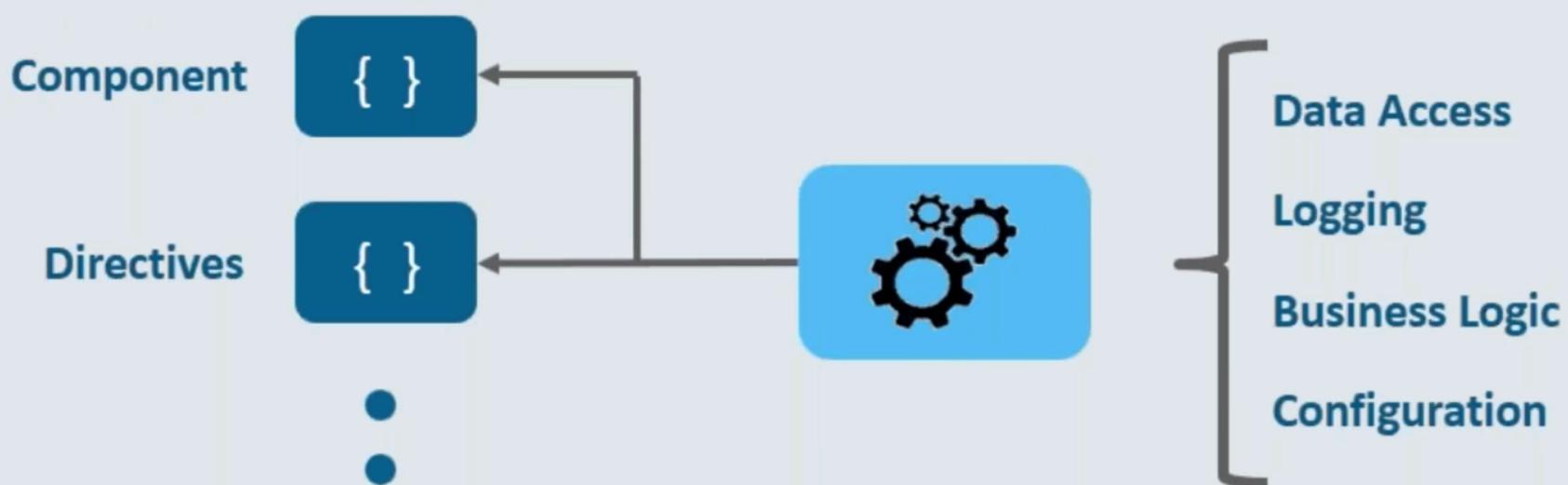


Service

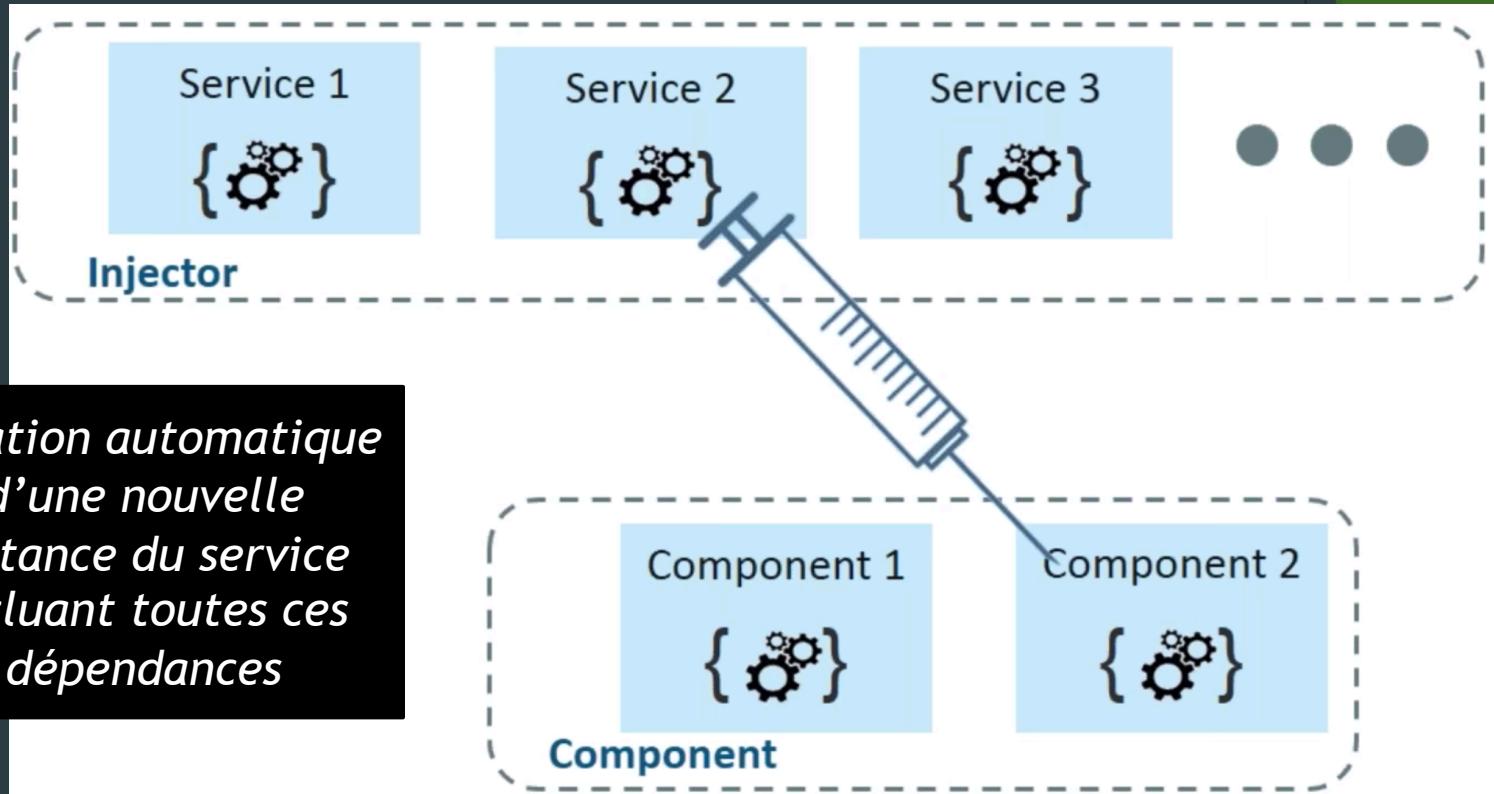
Service

- ▶ Morceau de logique partageable (injectée) dans l'application.
- ▶ Quand utiliser les services ?
 - ▶ Pour toutes logiques non visuelles comme un appel service REST, un calcul...
- ▶ Injection possible à plusieurs endroits
 - ▶ Dans le module: Assure que le service soit singleton (une instance pour l'application)
 - ▶ Dans le component: Assure de créer une instance spécifique pour le component en question et ces enfants

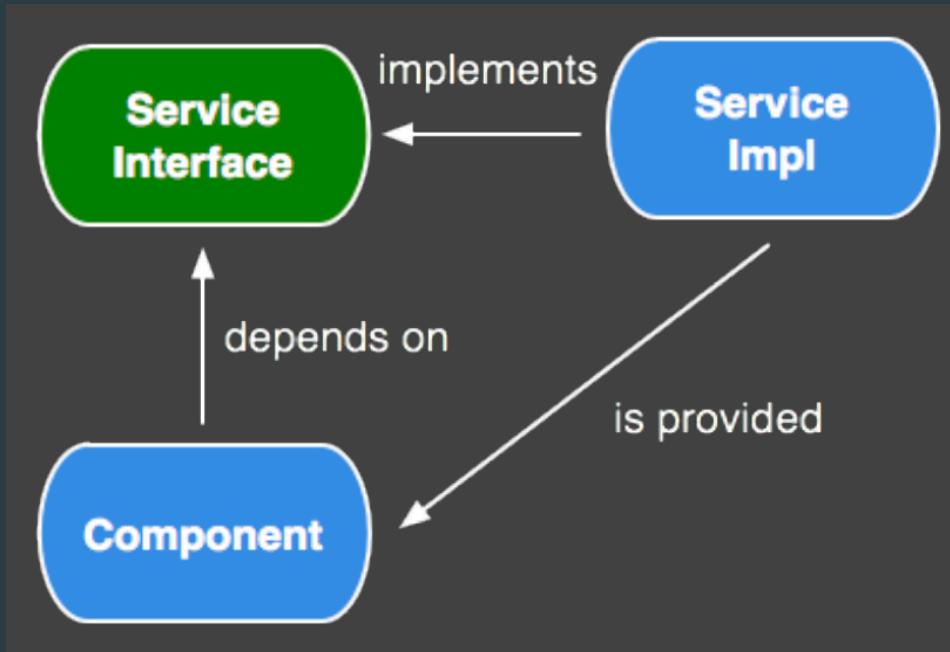
Service



Injection de dépendance



Injection de dépendance



Service



```
@Injectable()  
export class MyService {  
    ...  
}
```



Service

```
constructor(private myService: MyService) {}

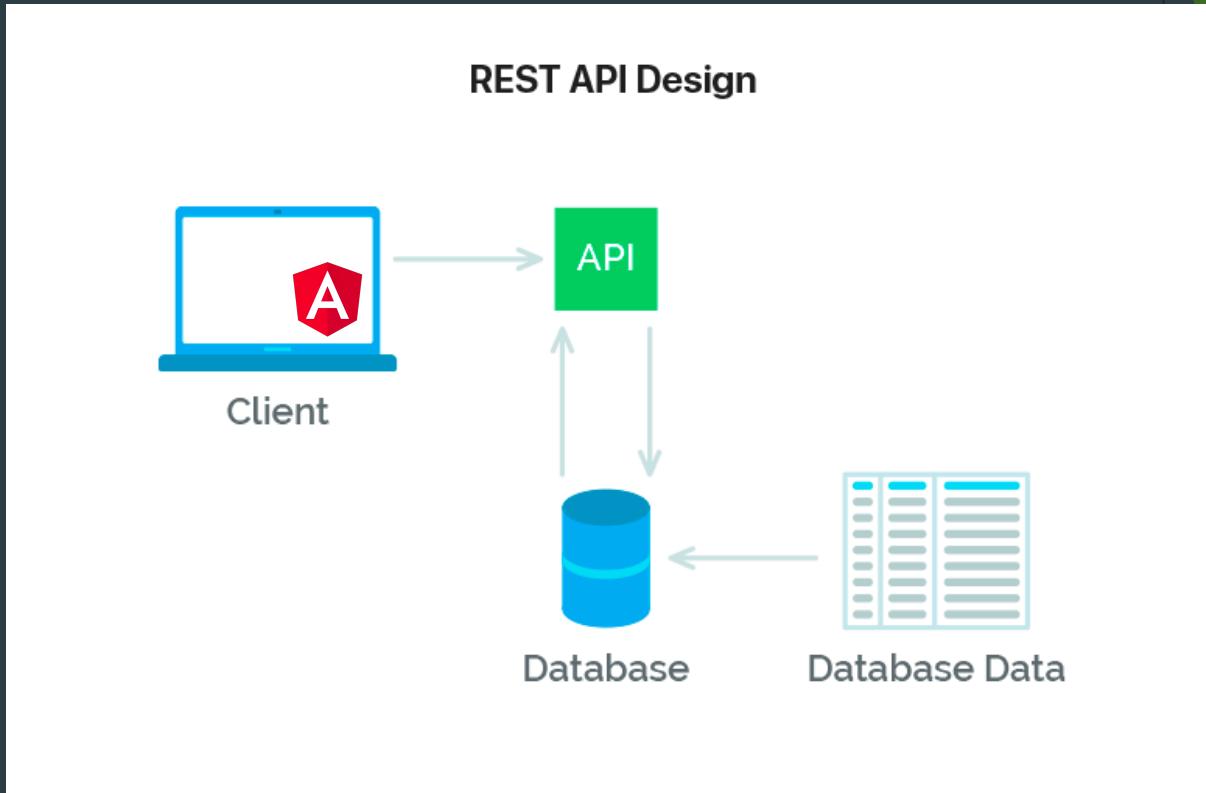
ngOnInit() {
  this.myService.myFunction();
}
```

Exercice

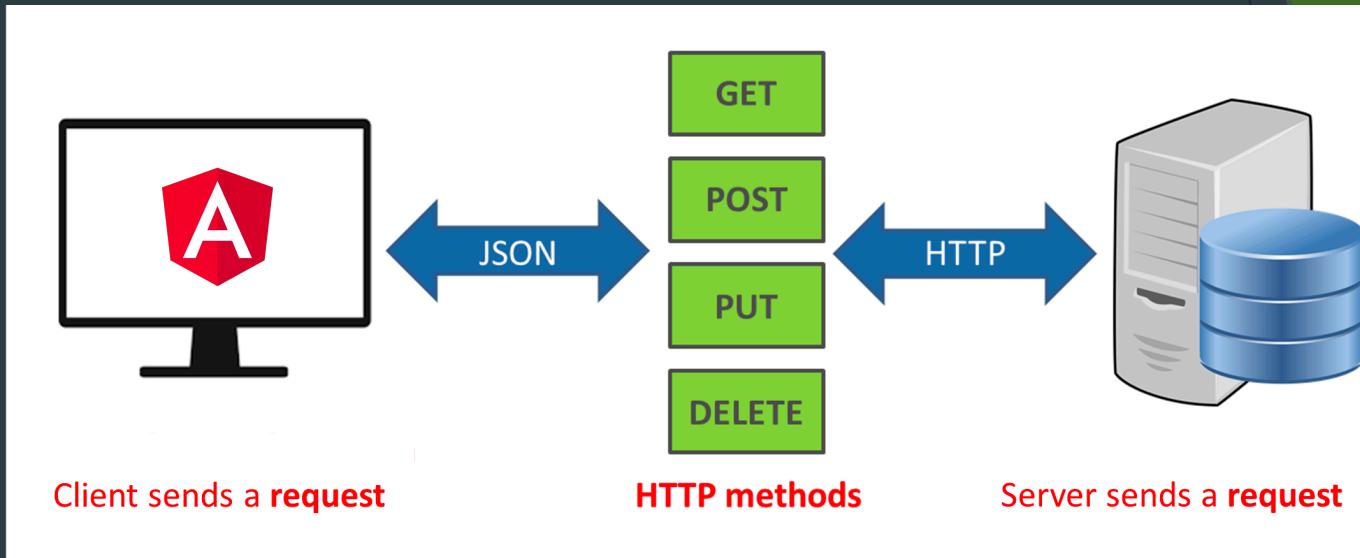
Service

- ▶ Créons le service « AuthenticationService » qui contient un observable « isConnected » provenant d'un sujet privé au service.

Service HTTP



Service HTTP



Service HTTP

```
@Injectable()
export class BookService {

    private bookListUrl =
        'https://www.googleapis.com/books/v1/volumes?q=extreme%20programming';

    constructor(private httpClient: HttpClient) {}

    public getBooks() {
        return this.httpClient.get(this.bookListUrl);
    }
}
```

Exercice

Service HTTP

- ▶ Cr ons un client pour une API de recherche sur Youtube



Formulaire

Formulaire

- ▶ Demander et récupérer de données fournies par un utilisateur

Name:

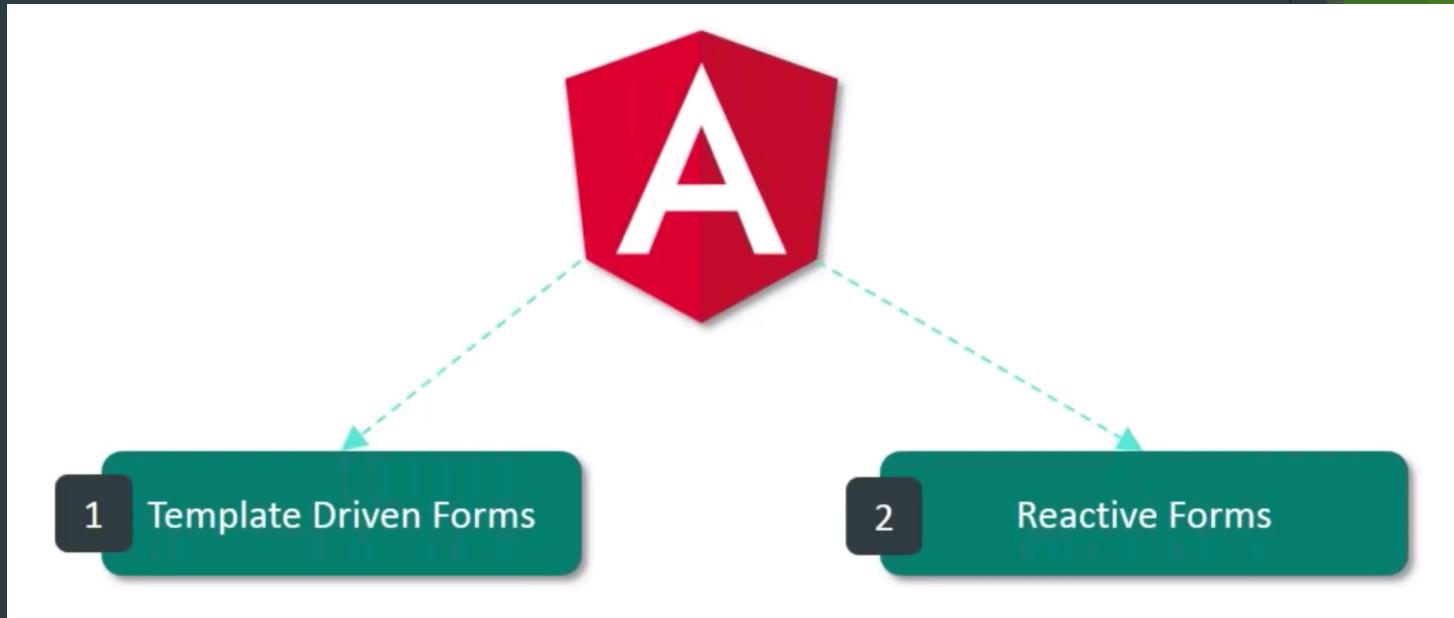
Contact:

Email:

Address:

Types de formulaire

- ▶ Il y a deux types de formulaire :





Template driven Formulaire

Template driver

Formulaire

Name:

Contact:

Email:

Address:

Submit

Template



Directives

Template driver

Formulaire

- ▶ Gestion du binding des contrôles input à l'aide de la directive [(ngModel)]
- ▶ Définition de validation à l'aide de directives
- ▶ Accès aux données du formulaire (statuts de validation) à l'aide d'une variable de template
 - ▶ #myInput="ngModel"
- ▶ Mutation



Template driver

Formulaire

```
<form #myForm="ngForm">

  <input type="text" id="myInput" name="myInput" #myInput="ngModel"
    required [(ngModel)]="myModel.myVariable"/>
  <span *ngIf="!myInput.valid && myInput.dirty">Requis !</span>

  <input type="email" id="myInput2" name="myInput2" #myInput2="ngModel"
    required [(ngModel)]="myModel.myVariable2"/>
  <span *ngIf="myInput2.errors?.required && myInput2.dirty">Requis !</span>

</form>
```

Exercice

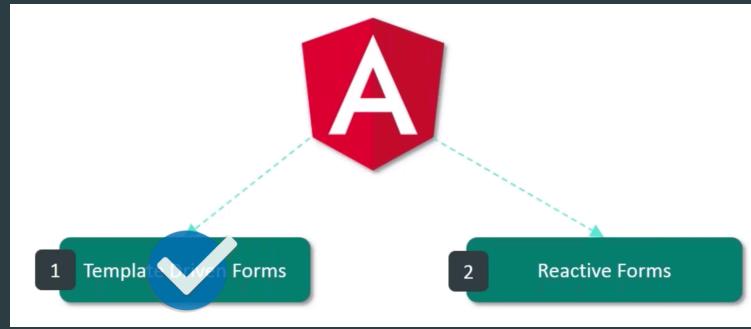
Template driven

- ▶ Cr ons un formulaire pour notre page de contact

Exercice

Template driven

- ▶ Cr ons une directive de validation custom



Reactive Forms

Formulaire

Reactive Forms

Formulaire

Name:

Contact:

Email:

Address:

Submit

Component
(TypeScript)



Reactive Forms
API

Reactive Forms

Formulaire

- ▶ Modèle dédié à l'HTML + Modèle business
- ▶ Création et manipulation d'un arbre de contrôles en TS qui comprend pour chaque contrôle:
 - ▶ La valeur de base du model dédié à l'HTML
 - ▶ La validation des contrôles
- ▶ Immutabilité

Reactive Forms

Formulaire

```
this.myForm = this.fb.group({
  myInput: this.fb.control(this.myModel.myVariable, [Validators.required])
});
```

```
public hasMyInputError() {
  const control = this.myForm.get('myInput');
  return control.errors && control.errors.required;
}
```

```
<form [formGroup]="myForm">
  <input type="text" id="myInput" name="myInput" formControlName="myInput"/>
  <span *ngIf="hasMyInputError()">Requis !</span>
</form>
```

Exercice

Template driven

- ▶ Copions la page de contact et adaptons-la pour fonctionner en Reactive Forms



Composant de formulaire

Formulaire

Composant de formulaire

Template Driven Forms
[(ngModel)]

Reactive Forms
formControlName

Form Component



Composant de formulaire

Built-in

- ▶ Input
 - ▶ Text
 - ▶ Checkbox
 - ▶ Radio
 - ▶ ...
- ▶ Select
- ▶ TextArea

Composant de formulaire

Custom

- ▶ Pour ce faire il faut:
 - ▶ Créer un composant
 - ▶ Implémenter des interfaces:
 - ▶ **ControlValueAccessor** : gestion de la valeur du control
 - ▶ **Validator** : validation custom
 - ▶ Spécifier à Angular que le composant est un ValueAccessor et/ou un Validator

Value accessor

Composant de formulaire - Custom

► Fonctions à implémenter :

- ▶ `writeValue(...)` : récupérer la valeur depuis le formulaire
- ▶ `registerOnChange(...)` : enregistrer le callback à appeler quand il y a un changement de valeur à transmettre au formulaire
- ▶ `registerOnTouched(...)` : enregistrer le callback à appeler pour informer le formulaire que votre composant a été touché
- ▶ `setDisabledState(...)` : récupérer l'état de disponibilité depuis le formulaire

Value accessor

Composant de formulaire - Custom

```
export class MyComponent implements ControlValueAccessor {  
  
    public value: number;  
  
    writeValue(value: number): void {  
        this.value = value;  
    }  
  
    registerOnChange(fn: any): void {  
        this.propagateChanges = fn;  
    }  
  
    registerOnTouched(fn: any): void {  
    }  
  
    private propagateChanges = (value: number) => { };  
}
```

Value accessor

Composant de formulaire - Custom

```
@Component({
  providers: [
    {
      provide: NG_VALUE_ACCESSOR,
      useExisting: forwardRef(() => MyComponent),
      multi: true
    }
  ]
})
export class MyComponent implements ControlValueAccessor {
```

Exercice

Composant de formulaire

- ▶ Cr ons notre premier composant de formulaire oui/non

Validator

Composant de formulaire - Custom

- ▶ Fonctions à implémenter :

- ▶ **validate(...)** : retourne null s'il n'y a pas d'erreurs, sinon retourne un dictionnaire d'erreurs



Validator

Composant de formulaire - Custom

```
export class MyComponent implements Validator {  
  
    validate(c: FormControl): { [key: string]: any } {  
        let result = null;  
  
        const value = this.form.value;  
        const currentDate = moment(` ${value.year}-${value.month}-${value.day}`);  
  
        if (!currentDate.isValid()) {  
            result = {  
                format : true  
            };  
        }  
  
        return result;  
    }  
}
```



Validator

Composant de formulaire - Custom

```
@Component({
  providers: [
    {
      provide: NG_VALIDATORS,
      useExisting: forwardRef(() => MyComponent),
      multi: true
    }
  ]
})
export class MyComponent implements Validator {
```

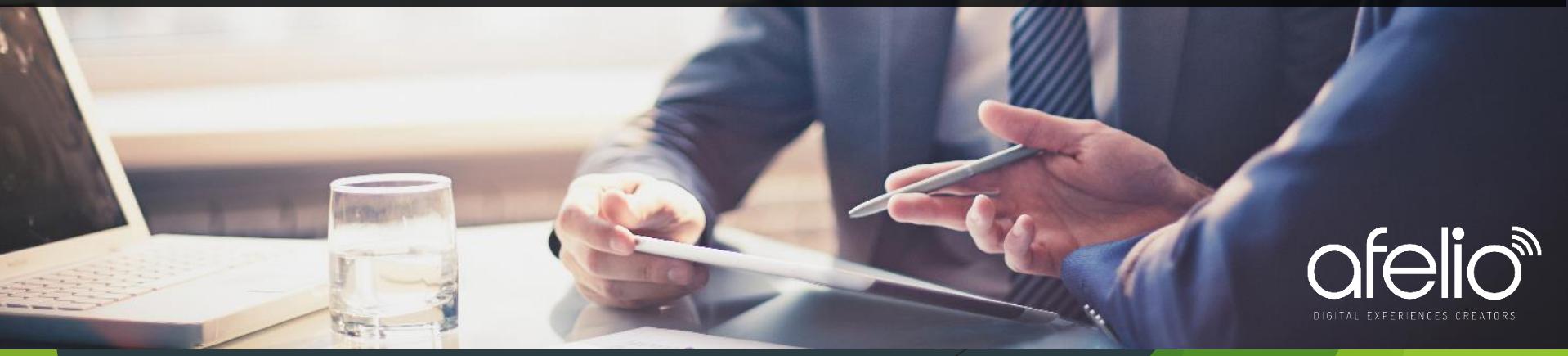
Exercice

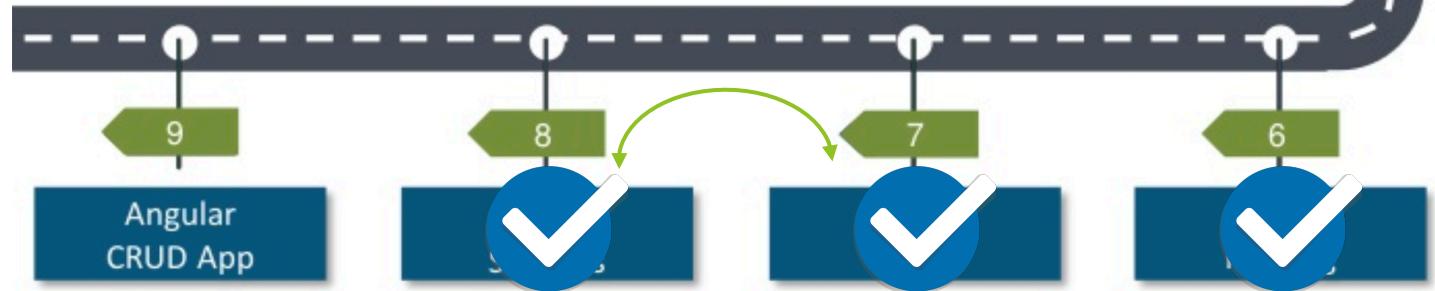
Composant de formulaire

- ▶ Ajoutons une validation à notre composant de formulaire

Thank You

Fin de la théorie





Références

- ▶ <https://www.youtube.com/watch?v=R4wGCHzn6-Q&list=PL9ooVrP1hQOF4aDuqaWYWStj1isPF6HHg>
- ▶ <https://vsavkin.com/the-core-concepts-of-angular-2-c3d6cbe04d04>
- ▶ <http://gilsdav.wetry.eu>